Huffman Coding Project

TOR 3 project

Gregor Antonaz, Miha Sivka

20.5.2025

Git repo: https://github.com/GorGre14/hauffman_code

Project description

- Restrict your alphabet to 6 letters {a, b, c, d, e, f} and construct the sequence of 300 symbols e.g. abacefddcabbdedacffaeebdfcabeafbdcab
- using the following probability distribution :
  Pb(a)=0.05, Pb(b)=0.1, Pb(c)=0.15, Pb(d)=0.18, Pb(e)=0.22, Pb(f)=0.3
- Huffman coding assigns shorter codewords to more frequent symbols and longer ones to less frequent symbols, thereby minimizing the expected codeword length
- In this project we show Huffman coding and decoding of the stream and compute the average length of codewords
- And what happens if the probability is changed so that Pb(a)=0.3 and P(f)=0.05

Code explanation

- Node class - defines nodes in the huffman tree
- Huffman_code – buildes a Huffman code for the given dictionary and probabilities
- Encoding – encodes the sequence with Huffman codebook
- Decoding – decodes the string back to its original form
- Expected_length – calculates the expected codeword length
- Averaged_realised_length – computes the empirical average codeword length
- Compression_ratio – estimates the compression ratio

Defining the probabilities and generating the sequence

```python
ALPHABET = ["a", "b", "c", "d", "e", "f"]

def main():
    random.seed()
    p_original = dict(zip(ALPHABET, [0.05, 0.10, 0.15, 0.18, 0.22, 0.30]))
    p_shifted  = dict(zip(ALPHABET, [0.30, 0.10, 0.15, 0.18, 0.22, 0.05]))


    sequence = random.choices(ALPHABET,weights=[p_original[s] for s in ALPHABET], k=1000)
```

Constructing the codebook and calculating its stats

```python
def run_experiment(k: int, probs: Dict[Symbol, float], seq: List[Symbol], header: str):
    if k == 1:
        prob_k = probs
        blocks = seq
    else:
        prob_k = {x + y: probs[x] * probs[y]
                  for x, y in itertools.product(ALPHABET, repeat=k)}
        blocks = ["".join(seq[i:i + k]) for i in range(0, len(seq), k)]

    code_k = huffman_code(prob_k)

    print(f"\n{header}")
    print("─" * len(header))
    print(f"Alphabet size       : {len(prob_k):>3}")
    H = entropy(prob_k)
    E_L = expected_length(code_k, prob_k)
    L_bar = average_realised_length(code_k, blocks)
    ratio = compression_ratio(code_k, blocks, k)
    print(f"Entropy             : {H:.4f} bits")
    print(f"Expected length     : {E_L:.4f} bits")
    print(f"Average length      : {L_bar:.4f} bits")
    print(f"Compression ratio   : {ratio:.4f}")
    return code_k, blocks
```

Encoding and decoding

```python
def encode(sequence: Iterable[Symbol], codebook: Dict[Symbol, str]) -> str:
    return "".join(codebook[s] for s in sequence)


def decode(bitstring: str, codebook: Dict[Symbol, str]) -> List[Symbol]:
    rev = {v: k for k, v in codebook.items()}
    decoded: List[Symbol] = []
    w = ""
    for bit in bitstring:
        w += bit
        if w in rev:
            decoded.append(rev[w])
            w = ""
    if w:
        raise ValueError("codebook not prefix-free?")
    return decoded
```

Defining the block size and probability shift and running everything

```
# block length k = 1
code1, _ = run_experiment(1, p_original, sequence,"Block length k = 1 (optimal code)")

# block length k = 2
code2, blocks2 = run_experiment(2, p_original, sequence,"Block length k = 2 (optimal code)")

# probability shift
print("\nProbability shift:  P(a)=0.30, P(f)=0.05")
E_L1_shift = expected_length(code1, p_shifted)
ratio1_shift = math.log2(len(ALPHABET)) / E_L1_shift
print(f"k=1 □  Expected length    : = {E_L1_shift:.4f} bits,   ratio = {ratio1_shift:.4f}")
prob2_shift = {x + y: p_shifted[x] * p_shifted[y]
               for x, y in itertools.product(ALPHABET, repeat=2)}
E_L2_shift = expected_length(code2, prob2_shift)
ratio2_shift = math.log2(len(ALPHABET) ** 2) / E_L2_shift
print(f"k=2 □  Expected length    : = {E_L2_shift:.4f} bits,   ratio = {ratio2_shift:.4f}")
#print("The sequence encoded: ",encode(sequence, code1))
```

Output

```
Block length k = 1 (optimal code)
---------------------------------
Alphabet size      :   6
Entropy            : 2.4058 bits
Expected length    : 2.4500 bits
Average length     : 2.4820 bits
Compression ratio  : 1.0415

Block length k = 2 (optimal code)
---------------------------------
Alphabet size      :  36
Entropy            : 4.8116 bits
Expected length    : 4.8456 bits
Average length     : 4.9180 bits
Compression ratio  : 1.0512

Probability shift:  P(a)=0.30, P(f)=0.05
k=1  -  Expected length    : = 2.9500 bits,   ratio = 0.8763
k=2  -  Expected length    : = 6.1031 bits,   ratio = 0.8471
```

Discussion

- When the symbol probabilities are suddenly changed so that P(a) = 0.30 and P(f) = 0.05, but you continue using the original Huffman code (which was optimized for P(a) = 0.05 and P(f) = 0.30), the compression ratio drops significantly
- Huffman coding is optimal only when the code matches the source distribution
- The compression ratio drops to ~ 0.87, meaning the method becomes inefficient and counterproductive